View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Trepo - Institutional Repository of Tampere University

TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

SEYED ABOLFAZL HOSEINI

EVALUATION OF DCI ARCHITECTURAL STYLE FOR FEATURE-ORIENTED SOFTWARE DEVELOPMENT

Master's thesis

Examiner: Professor Kari Systä
Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 9th January 2013.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Degree Programme in Information Technology
HOSEINI, SEYED ABOLFAZL: Evaluation of DCI Architectural Style for Feature-Oriented Software Development
Master of Science Thesis, 54 pages
February 2014
Major subject: Software architecture
Examiner: Professor Kari Systä
Keywords: Software architecture, Data Context Interaction, feature-oriented software development, DCI

Developers often spend large amounts of time working on implementing complex functions and features. Increment of the implemented features will result in growth of the complexity. Therefore, it is important to restrict the list of functional requirements to those that have value to the user or client and to ensure that requirements are phrased in a language that the user or client can understand. Object-Oriented Programming (OOP) was expected to help programmers to decrease the complexity by introducing class and object concepts, but the hope is not totally realized.

A new approach called Data-Context-Interaction (DCI) has been proposed as a solution. Thus, in this thesis I study DCI in order to understand if DCI can effectively and efficiently support Feature-Oriented Software Development (FOSD) in terms of adding a new feature and removing an old feature. The main goal of this work was to investigate the feasibility of DCI for software development practice where new features can flexibly added and removed. Through a comprehensive literature study and implementation of messaging example in both JavaScript and C#, my research has concluded that DCI is a useful paradigm but it is not applicable for all programming languages. Furthermore, it is not a suitable paradigm for all the possible use case scenarios.

PREFACE

This thesis concludes my Master of Science education in Information Technology at the Tampere University of Technology (TUT) in Tampere, Finland. The thesis is a pre-study for the DEBLOAT project proposed by Professor Kari Systä.
DEBLOAT aims to develop technologies in order to discover and remove unnecessary features of software in a systematic way.

I would like to thank Professor Kari Systä for his kind mentoring, without his supervisions and supports it was almost impossible to do this research. And I would like to thank my parents and friends who helped me with their mental supports.

SEYED ABOLFAZL HOSEINI
January 2014

CONTENTS

ABBREVIATIONS

AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
DCI	Data-Context-Interaction
ER	Entity Relationship
FOP	Feature-Oriented Programming
FOSD	Feature-Oriented Software Development
MVC	Model-View-Controller
NIAM	Nijssen's Information Analysis Methodology
ODMG	Object Data Management Group
OOP	Object-Oriented Programming
ORM	Object Role Modelling
TM	Transfer Money
UML	Unified Modelling Language

1. INTRODUCTION

1.1. Introduction

Rapidly changing technologies have established larger and more complex software. Moreover, new challenges are outcomes of distributed and large-scale development (Magdaleno et al. 2012). Systems and features get complicated with complex user requirements. The major problem is that the functional requirements mix the domain logic including: UI, data layer, and network communication functions with business logics. The result is that developers put lots of time and effort working on the technical features comparing to business characteristics. "A project that delivers a system with the greatest persistence mechanism but no business features is a failure". This problem can end up producing a system that does not do what the client requires (Stephen and Felsing 2001).

A good solution to this problem is to restrict the lists of functional requirements to those that are valuable to the user (Stephen and Felsing 2001). It is crucial for organisations to perform software maintenance in a way that keep the complexity arising from changes minimal, and reduce the potential for new bugs to be introduced by those changes. Hence, software developers need a comprehensive solution to help them understand changes and their impacts. This understanding has a significant importance because, as changes are made, architectural complexity increases. This will most probably result in an increase in the number of bugs introduced (Williams 2010).

Today's software development challenges includes multiple requirements and different domains that software must consider. Solutions suggested for these problems are for example multi paradigm design, Aspect-Oriented Programming (AOP), and Feature-Oriented Programming (FOP) (Günther and Sunkle 2012).

Consequently, the concept of Feature-Oriented Software Development (FOSD) arises. FOSD decomposes a software system in terms of the features and it provides flexibility. Thus, the elementary idea of FOSD is to decompose a software system to its features. "The goal of the decomposition is to construct well-structured software that can be tailored to the needs of the user and the application scenario" (Apel and Kästner 2009). Furthermore, FOSD is a paradigm designed for construction, customisation, and synthesis of large-scale software systems. Moreover, the concept of a feature is placed at the core of FOSD (Apel and Kästner 2009).

Today's OOP languages aim to focus on data as the primary organising structure. The main building blocks of these languages are the classes which contains the data and its local behaviours. Object orientation should develop further in order to provide the classes with the data and their related behaviours in overall structure of the program. So the data

will be increasingly encapsulated and hidden. This refocus will reduce the preoccupation with data and its local behaviours which was more related to the programmer organisation of code than to the business function or the end user benefits from the code (Coplien 2012). A new approach called Data-Context-Interaction (DCI) is proposed as a solution. The DCI paradigm facilitates the building of domain architecture and allows us to successfully decouple between domain and business-logic features. It also provides an approach for lean practitioners to design system architecture much closer to the real-world domain (Hayata et al. 2012).

In addition, DCI is an architectural paradigm to be used in conjunction with programming to construct a system of communicating objects. The goal of DCI is to make code more readable by focussing on the system behaviours and avoiding the fragmentation of the behaviours as typically seen in an object-oriented solution. This allows the rapidly changing system behaviour code to be developed and maintained independently of the slower evolving domain model (data classes). This also enables programmers to reason directly about the system-level state and behaviour rather than needing to create a map between their mental model, and the user's mental model – this leads to more easily maintainable code. Therefore, these characteristics make DCI very powerful, and make DCI a paradigm comparable with OOP, or Service-Oriented Programming (SOP) (Kutschera 2011).

In this thesis, I study DCI, FOSD and Aspect-Oriented Software Development (AOSD) paradigms. The purpose of this thesis is to research and use DCI in order to develop future-oriented software.

1.2. Purpose of the Thesis

The purposes of this thesis are:

1. To study DCI and build significant knowledge about DCI comprising Data, Context, and Interaction.
2. To mainly understand if the DCI paradigm can result in an effective and efficient feature-oriented software.
3. To briefly study FOSD.
4. To briefly study AOSD.

Consequently, to be able to meet the goals of this thesis, a research question is established in section 1.3, which will be answered in this thesis.

1.3. Research Question

The research question is:

> How well does DCI support flexible introduction of new features and removal of old features?

1.4. Structure of the Thesis

The structure as shown below (Figure 1.1), depicts the process through which the thesis is flowing. This thesis is divided into six different sections. Section one is the introduction to the thesis. In this section, I provide an introduction to the studied topic, the purpose of the thesis, the research question, and the structure of the thesis. Section two is the technology background. In section two, DCI, FOSD and AOSD is well discussed. Section three illustrates the researcher's hands-on implementation with both JavaScript and C# using the DCI paradigm. In section four, I present the results constructed from the literature studies and implementation. Section five covers the discussion. Finally, in section six I provide conclusions for the whole thesis. The picture below shows the thesis skeleton.

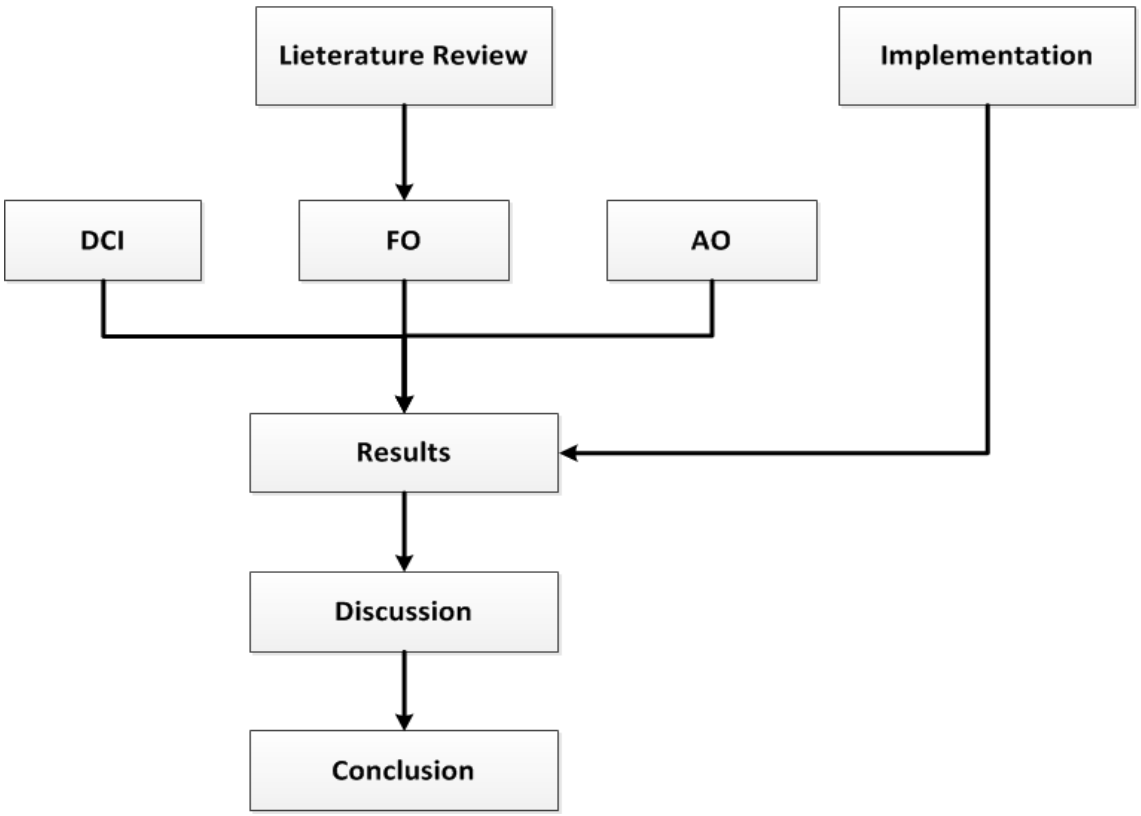

Figure 1.1. *Thesis skeleton*

2. TECHNOLOGY BACKGROUND

2.1. Introduction

In OOP, classes and objects are introduced. Analysing and developing a system based on objects and classes for a large software is complicated. Due to this complication, a system containing unused methods and attributes might be the result. On the other hand, there may be methods and attributes missing, which are needed. A system is driven and tracked by its functions. A user's sets of functional requirements are given to developers who implement the system. So, the system should provide the functionalities that clients have asked. If this goal is not achieved, the system has failed (Stephen and Felsing 2001).

With OOP the major problem is that the developers often spend large amounts of time working on implementing complex functions and features. A good solution to this problem is to restrict the lists of functional requirements to those of value to the user or client and to ensure that requirements are phrased in language that the user or client can understand (Stephen and Felsing 2001).

Hence, a user or client's requirements are expressed as features. Therefore, features are those requirements that users can understand while interacting with the system.

In this research, I study DCI, FOSD, and AOSD, which are the three paradigms that can be used in the design, development and programming phase of software development to ensure the production of feature-driven software. The reason why I study these three paradigms is that all three paradigms concentrate on a concept with a higher modularity level than the class objects and they all construct a system based on its features. In addition, user's mental model is considered by all of the paradigms.

The rest of this section is organised as follows: Section 2.2 explains the DCI paradigm; Section 2.3 describes FOSD; and finally in section 2.4, AOSD is explained.

2.2. Data Context Interaction

This section describes DCI in a broad way and concepts in DCI, mental model, overview of DCI, Model-View-Controller (MVC) and DCI and MVC are explained.

2.2.1. Overview of DCI

DCI is a paradigm used in software development to construct systems of communicating objects (Coplien and Bjørnvig 2010; Reenskaug 2008). Its goal is to make code more readable (Reenskaug and Coplien 2009; Reenskaug 2008). According to Reenskaug (2008), DCI stands for:

- ***D*** for ***Data:*** Data is constructed from the unification of the domain classes. (Reenskaug 2008).
- ***C*** for ***Context:*** Context maps all the roles to data objects and runs when a network of communicating objects executes a System Operation. Different objects may lead to different executions of the same operation. (Coplien and Bjørnvig 2010; Reenskaug and Coplien 2009; Reenskaug 2008).
- ***I*** for ***Interaction:*** Interaction includes methods that specify how objects work together (Reenskaug 2008).

DCI separates the program into different perspectives. Each perspective's focus is on certain system properties.

- Data perspective: the representation of system state based on data objects.
- Context perspective: the runtime networks of interconnected objects.
- Interaction perspective: how the networked objects collaborate to produce system behaviour (Reenskaug 2008).

The basis of DCI is the concept of roles with the Context. The roles are defined in Interaction. The responsibility of Context is to do the mapping of some roles onto some concrete data objects. The only responsibility of data objects is to access the data. Context is populated in response to user interactions and then the use-case is executed using this Context (Emoet 2011).

DCI meets the following goals (Reenskaug and Coplien 2009; Reenskaug 2008):

- To improve the readability of object-oriented code;
- To separate the code related to the system behaviour (what the system *does*) from the code related to the domain knowledge (what the system *is*);
- To allow developers to think and develop software based on system-level state and behaviour instead of only object state and behaviour;
- To support an object style of thinking that is close to user's mental models, rather than the class style of thinking that overshadowed object thinking early in the history of OOP languages.

The class definition in the current programming languages is probably the simplest way of describing the Data. The Data specifies the local behaviour of the objects. Behaviour is realized in the Data object's encapsulation. "Encapsulated local behavior cannot interfere with the Interactions that implement the System Operations." (Reenskaug 2008)

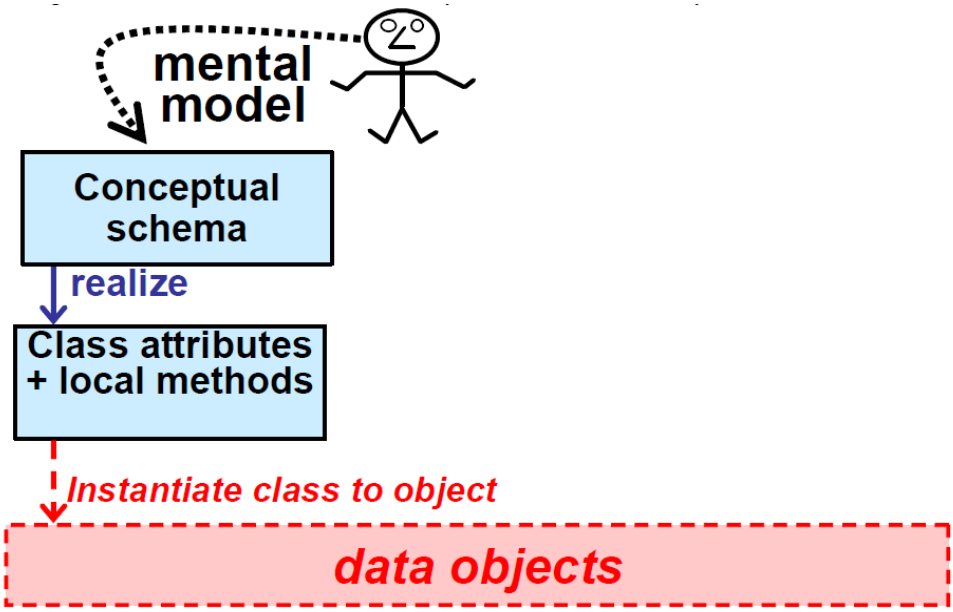

Figure 2.1. *User's mental model represented as state part of Data classes* (Reenskaug 2008)

Thus data classes are instantiated to form an object structure that corresponds to the user's Conceptual Schema (Reenskaug 2008). Conceptual schema is the representation of user's mental model by the use of some modelling techniques and languages. Conceptual schema acts as the communication tool among designers, programmers, users and managers. We can name Entity Relationship (ER) modelling, Object Role Modelling (ORM), and Unified Modelling Language (UML) as the three main known conceptual schema modelling techniques (Campbell 1996). Reenskaug believes for DCI perspective, Nijssen's Information Analysis Methodology (NIAM), Object Data Management Group (ODMG), or simply UML class diagrams could be effective languages for schema (Reenskaug 2008).

A set of instantiated data classes that are structured according to the schema simply do an important task for the user and that is direct communication between the user's mental model and the system. The creation or removal of data objects and their relations are runtime incidents. In figure 2.1, the box with dashed border shows the runtime elements. The system's response to user commands is the system runtime behaviour. A user command triggers a method in one of the system's objects. This method sends additional messages in such a way that the commands are produced by a network of communicating objects (Reenskaug 2008).

"A program that follows the DCI paradigm exposes its inner workings to a reader of its code." (Reenskaug 2008) In DCI, each network of interconnected objects is coded as a corresponding network of the connected Roles. The responsibility of the participating data objects is not to establish and maintain the runtime network structure. This is centralised in Context. The system behaviour part of the DCI is shown in figure 2.2 (Reenskaug 2008):

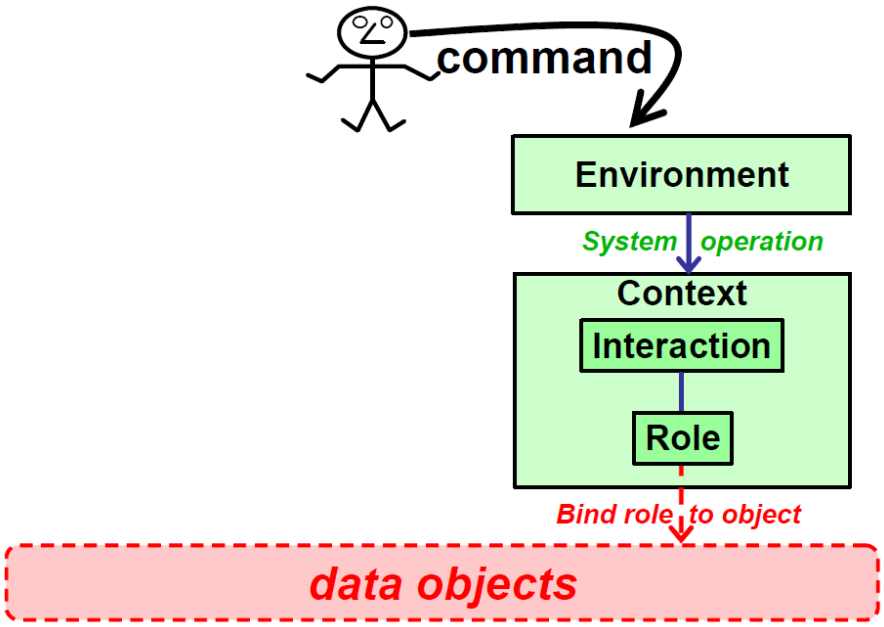

Figure 2.2. *System behaviour realised in a Context (*Reenskaug 2008*)*

Environment is a class that triggers execution of the system operation with sending a command to the Context.

System behaviour and Interaction messages are shown in figure 2.2. "[The] weakness is that the methods triggered by these messages will normally be properties of the data classes." (Reenskaug 2008) This actually means that different objects playing the same Role may handle the same message with different methods and violate the network topology specified in the Context (Reenskaug 2008).

To resolve explained problem, all objects playing a given Role should process the same Interaction messages with the same methods. These kinds of methods are called Role Methods. Role Methods are the functionalities of the Role. "The Role Methods are injected into the data classes that may not override them." (Reenskaug 2008) The reader of the code can establish with certainty that there are no hidden issues in the details of the data classes (Reenskaug 2008).

Complete representation of DCI with its relations from Role to class is illustrated in figure 2.3. The relationship means that the instances when these classes will give priority to the Role Methods above any methods is defined in the class itself (Reenskaug 2008).

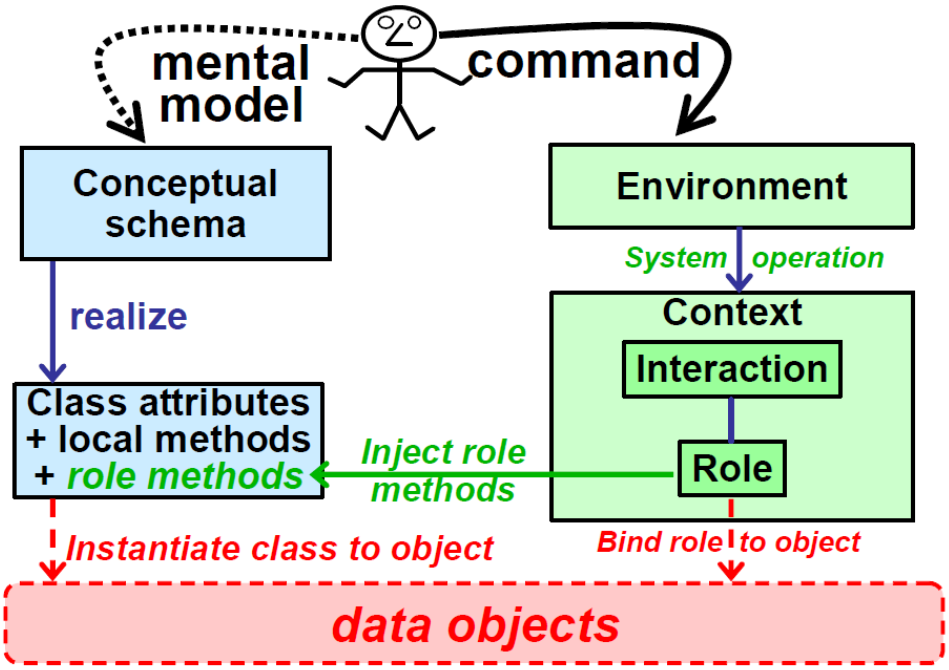

Figure 2.3. *DCI paradigm with its complete relations (*Reenskaug 2008*)*

Aligning the data model of program with the user's mental model is the main objective of OOP. This alignment is the key to a good human computer interaction and better user experience (Coutaz 2006; Emonet 2011). OOP in its current state has not successfully accomplished this goal. Hence, DCI is considered as one practice to accomplish the goal. Last but not least, DCI plays a complementary role in relation to MVC (Coplien and Bjørnvig 2010). MVC is discussed in the following section.

2.2.2. Concepts in DCI

Below the most important concepts of DCI are listed in alphabetical order. See the list for the definition of unfamiliar keywords.

- **Command:** A user input that activates the execution of a System Operation (Reenskaug 2011).
- **Context:** A Context maps all the Roles to Data objects (Reenskaug 2011). The purpose is to bind the roles to some existing objects in order to have some RolePlayers for the defined Roles to perform the scenario. The Context finds needed objects to play each role, and objects learn from the Context how to play their roles in the use cases. This process is done with injecting the role's methods to the objects by the Context (Coplien 2010).
- **Data:** The Data is the user's mental model (Reenskaug 2003) and it consists of data classes (domain model).
- **Interaction:** Interaction includes methods that specify how objects work together and how a network of communicating objects realises a System Operation (Reenskaug 2011; Reenskaug and Coplien 2009).

- **Scenario:** Scenario is a storyline that describes user interactions and the system. Information about goals, expectations, motivations, actions and reactions (Coplien and Bjørnvig 2010) are included in the scenario.
- **System Operation:** System Operation is the behavior that is implemented by an Interaction and Context executes it. System Operation is at all times executed within a Context instance and is realised as an Interaction within this Context (Reenskaug 2011).
- **Trait:** Trait is a group of methods acting as a building block for classes in order to achieve better modularity and reusability. Other classes can use Trait without requiring multiple inheritances (Schärli et al. 2003).
- **Use Case:** Use case describes a series of interactions between a program and a user. Use case is therefore guiding the user towards a business goal. In a simple words, Use Case is a series of user interactions are described in a series of Scenarios (Reenskaug 2011).
- **Injection:** Injection means adding some functionality to an object in runtime**.** RoleMethod injection is a mechanism that maintains the invariant: "For any given Role, its RoleMethods are shared among all its RolePlayers." (Reenskaug and Coplien 2009; Reenskaug 2011).
- **Role:** Role is behaviour of an object in a network of interacting objects within a Context instance (Reenskaug 2011). We use roles to capture the main user concepts that participate in a Use Case requirement (Reenskaug and Coplien 2009). Each object has a role and a role identity and external properties of an object, while it ignores the object's internal construction (Reenskaug 2011). Roles are first-class components of the end user's cognitive model (Reenskaug and Coplien 2009). Role forms a bridge between the compile time and the runtime system properties. Role specifies an interface that should be implemented in all its RolePlayers (Reenskaug 2011).
- **RolePlayer:** RolePlayer is the object that fills the position of a Role in the communicating objects network (Reenskaug 2011).
- **RoleMethod:** RoleMethod is a stateless method where a feature of a Role is shared among all the Role's potential RolePlayers. RoleMethods have priority over methods specified in the RolePlayer instances. An executing RoleMethod can access actual parameters and temporary variables of RoleMethod. It can also access the current RolePlayer and the current Context (Reenskaug 2011).

2.2.3. Mental Model

Mental Model is defined as an explanation in someone's thought process for how something works in the real world. One can have number of mental models in her mind. Mental models can be created simultaneously and it is possible to switch between them when a need arises. Two aspects of mental models exist. First is the user's knowledge about how

to operate the system using commands and how to interpret the results. The other important aspect is for the user to understand the information model. Different tasks require different information models. The idea is that the system's data model might not be the same as the user's information model. The user can only communicate with the system data model through the user interface (Reenskaug 2011).

2.2.4. Model-View-Controller

Model-View-Controller (MVC) programming is an architectural paradigm to separate the software components to 3 parts: Model, View and Controller (Krasner and Pope 1988).

- **Model**: The Model is construct of objects of different classes which performs operations related to the application domain. Model does the actual work that is the simulation of the application domain (Krasner and Pope 1988).
- **View**: The View takes care of displaying the application's state and facets of the Models and it will be kept distinct from the Model. The Controller is mainly user interactions with the Model and the View (Krasner and Pope 1988).
- **Controller**: The Controller is an interface between the Model and its Views. A Model could have as many as View/Controller pair which is needed, but a View is closely connected to a single Controller and has just one Model (Krasner and Pope 1988).

MVC architecture is shown in figure 2.4. In MVC, the Controller maps the Model to the View (Curteanu 2010). This means that the mental model of the user is not directly connected to the Model (Reenskaug and Coplien 2009).

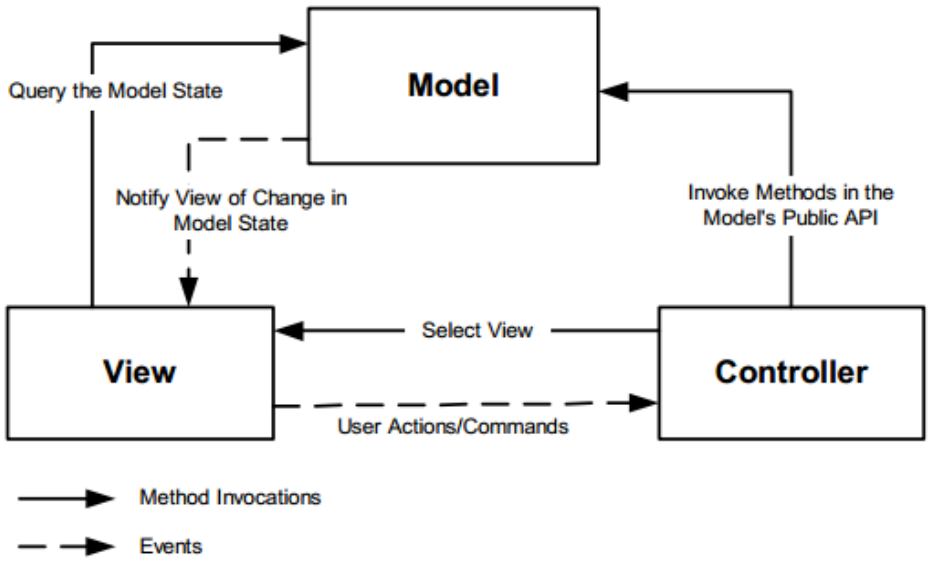

Figure 2.4. *MVC (Gulzar 2002)*

MVC does not directly cover the data access layer. This means that the model encapsulates the data. Therefore, the business logic and data in MVC is less coupled (Curteanu 2010).

2.2.4.1 MVC and DCI

DCI is a complementary for the MVC. The main goal of DCI is to result in a separation of the system state representation and system behaviour representation. This separation is related to goal of MVC - separation of data representation and user interaction - but in a same time different (figure 2.6). MVC and DCI are designed to facilitate programmers reasoning about the end user's mental models and capturing that in the code (Coplien and Bjørnvig 2010).

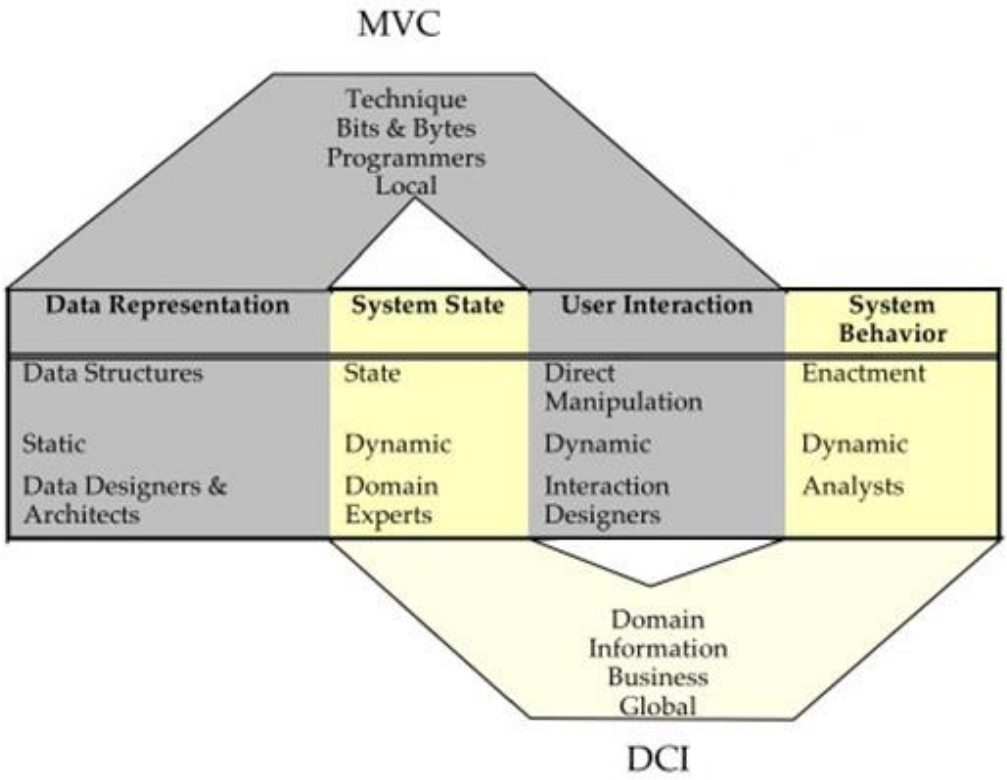

Figure 2.6. *Relationship between MVC and DCI (Coplien and Bjørnvig 2010)*

In DCI, the idea is to have a system that provides a short path from the information to the data model (Reenskaug and Coplien 2009). As shown in figure 2.5 the Roles and classes are designed in a way that reflect user's mental model. In addition, DCI can capture the end user cognitive model of roles and interactions between them (Qing and Zhong 2012).

Thus, as seen in figure 2.5, In DCI an object of a class supports not only the member functions of the class, but it can also execute the member functions of the role it plays at

any given time as though they were its own (Reenskaug and Coplien 2009; Reenskaug 2008). Figure 2.5 illustrates the sharing functions of MVC and DCI.

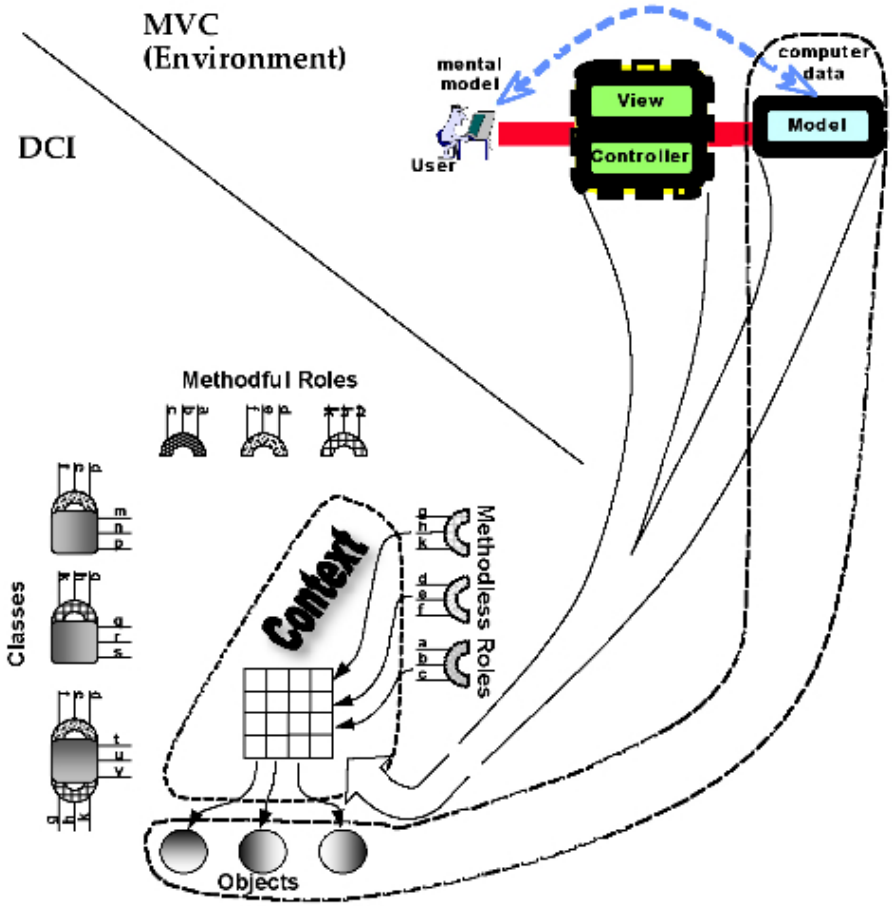

Figure 2.5. *MVC vs. DCI: mapping roles to objects (Reenskaug and Coplien 2009)*

In figure 2.5, it is clear that the Model and Controller initiates the mapping. The arrow from the Controller and Model into the Context can clearly show this initiation. Moreover, in the DCI part, we can consider Context as a table that maps role member functions in the rows of the table onto an object method placed in the columns of the table (Reenskaug and Coplien 2009). Reenskaug and Coplien (2009) concluded that MVC and DCI are the two paradigms that are designed for programmers to effectively implement the program by incorporating both paradigms in their design, development and programming.

2.2.5. Advantages of DCI

Below the most essential advantages of DCI are listed.

- DCI is a natural fit for Agile software development (Hayata et al. 2012; Reenskaug and Coplien 2009);
- DCI improves the readability of object-oriented code (Reenskaug 2008; Hayata et al. 2012);

- DCI separates the code related to the system behaviour (what the system *does*) from the code related to the domain knowledge (what the system *is*) (Coplien 2010);
- DCI reduces the size of the inheritance tree of the domain model, when compared to OOP (Reenskaug 2008);
- DCI makes reading the code more like the users' mental model when compared to a services model solution, and helps developers communicate with users' mental model (Coplien 2010; Reenskaug and Coplien 2009);
- DCI can be combined with frameworks or application servers that consider cross-cutting concerns like transactions and security, as well as other concerns like resource management, concurrency, scalability, robustness and reliability (Reenskaug and Coplien 2009);
- DCI achieves the goal of lean practices that enable and track consistency (Hayata et al. 2012);
- The DCI approach allows developers to clearly separate domain (what the system is) and business logic (what the system does) features (Hayata et al. 2012);
- DCI allows users an experience of working directly with his/her mental model when working with the computer (Reenskaug 2011);
- DCI ensures system lifecycle shortening and expensive reinvestment into a new platform (Reenskaug and Coplien 2009);

2.2.6. Summary

DCI is a paradigm used in software development to construct systems of communicating objects. Its goals are to make code more readable by avoiding the fragmentation of the behaviours as typically seen in an object-oriented solution.

- **Data:** Data includes the system domain classes.
- **Context:** When a System Operation is executed by a network of communicating objects, Context maps the object of Data to the Roles in Interaction.
- **Interaction*:*** Interaction includes methods that specify how objects work together.

Model-View-Controller (MVC) programming is an architectural paradigm to separate the software components to 3 parts: Model, View and Controller.

- **Model:** Model is a business logics layer.
- **View:** View is responsible for forwarding the user request to the Controller.
- **Controller:** Controller accepts the user request and controls the business object to meet the user request.

MVC separates the parts that are responsible for representing the information in the system and the parts that are responsible for interaction with the user. Moreover, the concept of the users' mental model is used in both MVC and DCI. The mental model is the feeling

of the user about what they wish to do with the system. In other words, the mental model provides the user a fantasy way of working directly with their own mental model while working with the system. This is the goal of DCI. The goal of MVC and DCI is to give the user a fantasy way of working directly with his mental model while working with the computer.

2.3. Feature-Oriented Software Development

Software quality has been subject of many past pieces of research. As discussed earlier classes in OOP are insufficient to capture all features of the software in a modular way. As a result, the last decade has seen quite a number of approaches that concentrate on more appropriate representation of features in the source code. A distinctive characteristic of these approaches is to focus on particular units of a software system such as files, classes, modules, components or binaries and how their technical attributes and patterns of change impact their quality (Cataldo and Nambiar 2012; Curtis 1986; Nagappan and Ball 2007).

Feature-Oriented Software Development (FOSD) is a paradigm for construction, customisation, combination of large-scale and variable software systems (Apel and Kästner 2009; Kästner and Apel 2013) and for designing and implementing applications based on features (Kästner et al. 2009) focusing on structure, reuse and variation (Kästner and Apel 2013). Moreover, FOSD encourages the systematic application of the feature concept in all phases of the software lifecycle (Kästner and Apel 2013).

Besides, FOSD is a conglomeration of different ideas, methods, tools, languages, formalisms, and theories but it is not just a single development method or technique. The system features connect all these developments (Apel and Kästner 2009). In general, a feature is an end-user visible characteristic (Kästner et al. 2009). In other words, a feature is a general requirement of stakeholders (Günther and Sunkle 2012) or a requirement in a software system (Kästner et al. 2009). According to Kavand et al. (2011), the main goal of FOSD is to increase reusability in terms of reusing features (Kavand et al. 2011).

The basic idea of FOSD is to decompose a software system in terms of the features it provides (Apel and Kästner 2009). The goal of decomposition is to construct well-structured variants of the software, which can be tailored to the needs of the user and the application scenario (Apel and Kästner 2009; Kästner and Apel 2013). FOSD also shares goals with other software development paradigms such as stepwise and incremental software development (Parnas 1976; Wirth 1971), AOSD (Batory et al. 2005), and component based software engineering (Szyperski 2002; Kästner and Apel 2013).

FOSD includes all phases of software development based on software features. Feature-Oriented Programming (FOP) is a major subset of FOSD, which will be briefly discussed in the following section.

2.3.1. Feature-Oriented Programming

FOSD is consist of the complete cycle of software development i.e. Analyse, Design and Programming. Feature-Oriented Programing (FOP) is a technique to localise, separate, and modularise crosscutting concerns (Szyperski 2002). The concern is a concept or a point of interest in the code. System level concerns like error handling, debugging and authentication which cut other general concerns called crosscutting concerns.

The key idea of FOP is to decompose a system's design and code along the features it provides (Baxter and Mehlich 2001) and implement so-called orthogonal features such as Aspects (Filman 2004). FOP follows a disciplined language-oriented approach, based on feature composition (Kästner and Apel 2013). In FOP, a feature has properties needed to define the feature characteristics and the roles feature play in programming. Feature properties are seen as the defining steps when using FOP (Günther and Sunkle 2012).

Different combinations of feature modules satisfy different end-user requirements or application scenarios. A feature module refines the content of a base program either by adding new elements or by modifying and extending existing elements. The order in which features are applied is important; earlier features in the sequence may add elements that are refined by later features (Thanker 2007; Kästner and Apel 2013).

2.3.2. Summary

Many approaches concentrate on a more appropriate representation of features in the source code. A distinctive characteristic of these approaches is to focus on particular units of a software system such as files, classes, modules, components or binaries and how their technical attributes and patterns of change impact their quality.

A feature-oriented approach present the design of feature-oriented software based on the feature. Due to the diversity of research about FOSD, there are several definitions of a feature. However, in this thesis, a feature is a user requirement that should be effectively and efficiently implemented.

On the other hand, FOSD is a paradigm for construction, customisation, and synthesis of large-scale and variable software systems and for designing and implementing applications based on features. Moreover, FOSD encourages the systematic application of the feature concept in all phases of the software lifecycle.

2.4. Aspect-Oriented Software Development

Construction of complex software systems requires special approaches capable of managing complexity at every level of the development process (Platunov and Nickolaenkov 2012). Aspect-Oriented Software Development (AOSD) has emerged as a new modularity practice (Laddad 2003). This modularity principle allows the division of a complex system into a set of simple components. AOSD aims to separate the implementation of requirements and design elements that affect multiple modules which are defined as crosscutting concerns (Laddad 2003). This separation is the fundamental approach that

helps to isolate different parts of the complex system and enable the programmer to analyse them independently (Platunov and Nickolaenkov 2012).

Global properties and programming and design issues can lead to crosscutting concerns. Error handling or transaction code, interacting features, and reliability and security are such examples (Murphy 2006).

During recent years, increased use of AOSD techniques as a means to modularise crosscutting concerns in software systems has been witnessed. Numerous Aspect-Oriented Programming (AOP) frameworks exist. Such frameworks include AspectJ, JBoss, and Spring (Rashid et al. 2010; Kiczales et al. 2001). Developers use AspectJ for crosscutting concerns as logging system operations, debugging, and coding recurrent features. AOSD includes all phases of software development based on aspects, and AOP is the most important subset of AOSD. This is briefly discussed in the following section.

2.4.1. Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a technology and relatively recent approach in the software engineering and programming languages communities (Viega and Voas 2000, Madeyski and Szała 2007). The AOP approach is supposed to improve quality attributes such as modularity, readability and simplicity (Madeyski and Szała 2007). The reliability of programmes is increased by modularising error-handling policies and allowing for easier maintenance and better reuse (Laddad 2003).

AOP provides a new construct, an aspect, to modularise crosscutting concerns in code. There are two types of crosscutting concerns: dynamic and static. In dynamic crosscutting, behaviour is modified by augmenting or replacing the core program execution flow in a way that cuts across modules. In static crosscutting, modifications are made to the static structure of the system (Laddad 2003). However, for better support of crosscutting design decisions, AOP uses a component language to describe the basic functionality of the system, and aspect languages to describe the different crosscutting properties. An aspect weaver is then used to combine the components and aspects into a system (Murphy et al. 1999). The weaving process can have a significant effect on the semantics of a primary concern. Weaving process can introduce new data and have control on dependencies in the primary concern and alter or even reduce existing dependencies (Murphy et al. 1999; Roger 2003).

Normally, an Aspect encapsulates constructs such as Aspect Advice, Pointcuts, and Intertype declarations. Below these types are briefly explained.

- Aspects are typed entities with predefined functionality. In terms of classes, aspects are unlike classes in that they are meant to confine crosscutting concerns to be injected into other types. In addition, aspects can contain new programming elements that classes cannot (Viega and Voas 2000).
- Advice is a functionality that is executed when an exposed join point is reached. Advice can be specified as Before Advice, After Advice and Around Advice. Before Advice executes before the join point. After Advice executes after the join

point and Around Advice executes instead of the join point but can also execute the original join point. The aspects are woven into the primary code by a pre-processor, compiler, or runtime system (Hannemann and Kiczales 2002).

- Pointcut is a set of join points specified by a Pointcut expression (Hannemann and Kiczales 2002).
- Intertype declarations are the members which are declared by Aspects and can cut across the hierarchy tree of classes (Braak 2006).

Using aspects will result in a better understandability and maintainability of the application (Hannemann and Kiczales 2002; Zhang and Jacobsen 2003; Coady and Kiczales 2003; Kiczales et al. 1997). Maintenance of the legacy applications consumes more time and resources than any other part of the software lifecycle (Zagal et al. 2002) and therefore developers might want to adapt to the techniques that reduce maintenance costs. The other expected benefit of AOP is that implementation of crosscutting concerns shows a high degree of variability and using AOP would implement these concerns in a consistent manner and prevent the high level of variability (Bruntink 2007).

The application of AOP and DCI are similar in many ways like focusing on separating concerns. In line with the fundamental principles underlying AOP, Roles in DCI aggregate and compose well unlike Aspects. Contexts provide scoped closure of association for sets of roles, while Aspects can only pair with the objects to which they are applied (Reenskaug and Coplien 2009; Viega and Voas 2000).

2.4.2. Advantages of AOP

There are number of essential advantages of using AOP and they are listed below.

- AOP contributes to reuse because it improves modularity but it does not reduce lines of code (Key Miller 2001).
- By using AOP and aspects in design and implementation, programmers are able to maximise the production and accuracy in their codes (Lieberherr et al. 1988; Key Miller 2001; Murphy 2006).
- AOP programmers are able to write references to aspects at join points; the appropriate places in code where the aspects belong (Key Miller 2001; Murphy 2006).
- With AOP many lines of scattered code are eliminated. If not, programmers would have to spend a substantial amount of time writing, tracking, maintaining, and changing codes. This helps to modifying and upgrading applications (Key Miller 2001).
- AOP allows programmers to change an aspect once and then they can affect the aspect wherever it occurs in an application (Key Miller 2001).
- AOP can also be used for configurable programmes like platform-portable, multiple-functionality, mobile, or distributed applications (Key Miller 2001).

2.4.3. Summary

AOSD has emerged as a new modularity practice. This modularity principle allows the division of a complex system into a set of simple components. Thus, AOSD aims to separate the implementation of requirements and design elements that affect multiple modules which is defined as crosscutting concerns. This separation is the fundamental approach that helps to isolate different parts of the complex system, and it enables the programmer to analyse them independently.

The AOP approach is designed to improve features and functions of the system, such as modularity, readability and simplicity. The reliability of programmes is most often done by modularising error-handling policies and allowing for easier maintenance and better reuse. Moreover, AOP provides a new construct, an aspect, to modularise crosscutting concerns in code.

There are a number of outstanding advantages of using AOP. Using aspects will result in a better understandability and maintainability of the application. The other expected benefit of AOP is that implementation of crosscutting concerns shows a high degree of variability and using AOP would implement these concerns in a consistent manner and prevent the high level of variability. In terms of DCI, AOP applications can be met by DCI and many of the goals of aspects in separating concerns.

3. IMPLEMENTATION – CASE STUDIES

3.1. Transfer Money

Reenskaug uses mini-scenario of Transfer Money (TM) to give a concrete example of DCI. The mini-scenario is similar to the money transfer process in an ATM machine. If one were to think of ATM system implementation, the most common scenario would be transferring money. What is a user's experience of transferring money? What does the user say and how do they express this experience? The most common answer given by ATM users about what is important is, selecting the source account, selecting the desired function and in the case of transferring money: to write the amount to be transferred and eventually to select the destination account. In simple words, the design should reflect the mental model of the users.

The set of selections made by the user represents the user's mental model. This kind of expression is what the system should do. The possible use case scenarios (what the system is) for the TM are as follows:

- Account holder chooses to transfer money from one account to another
- System displays valid accounts
- User selects source account
- System displays remaining valid accounts
- Account holder selects destination account
- System requests amount
- Account holder inputs amount
- Moving the transferred money and doing accounting process

The possible set of steps for the algorithm (what the system does) are as follows.

- Source account begins transaction
- Source account verifies funds available (notice that this must be done inside the transaction to avoid an intervening withdrawal!)
- Source account reduces its own balance
- Source account requests that destination account increase its balance
- Source account updates its log to note that this was a transfer (and not, for example, simply a withdrawal)
- Source account requests that destination account update its log
- Source account ends transaction

Eventually, the source account informs the destination account that the transfer has been made successfully (Reenskaug and Coplien 2009).

3.1.1. JavaScript Implementation

The DCI implementation of Anders Nawroth (Nawroth 2009) has been studied. In this study, DCI is implemented as a separate code file in JavaScript. The DCI core that is implemented by JavaScript code file, can be easily added to the project by copying its file (*DCI.js*) to the project directory and including it in any code file want to use DCI architecture. Data, Context and Interaction are implemented for the TM.

3.1.1.1 Data

As we studied in section 2.2, The Data represents what the system is and it consist of some classes. TM implementation requires an account class to instantiate some objects that represent the different accounts, such as the Saving Account, Investment Account, Family Account, etc.

The Account class includes a balance field to hold the account balance and an availableBalance() method to return the amount of balance. In addition, methods withdraw() and deposit() are for withdrawing and depositing operations. UpdateLog() method is designed to send information about any money transfer operation to browser debugger console as a log. The lines of codes below illustrate the Data.

```
function Account( type, initialBalance)
{

        var balance = initialBalance;
        this.availableBalance = function()
        {
            return balance;
        };

        this.withdraw = function( amount )
        {

            if ( balance < amount )
            {
                log.error( type + ": Insufficient funds!" );
                return;
            }

            balance -= amount;
        };
        this.deposit = function( amount )
        {
            balance += amount;
        };

        this.updateLog = function(msg, date, amount )
        {
            log.warn( [ "Account: " +
            this, msg, date, amount ].join( ", " ) );
        };



        this.toString = function()
```

```
        {
            return type;
        };
    }
```

In JavaScript, class concept has no syntactical structure like some other programming languages and implementation of class is done using a function structure.

3.1.1.2 Interaction

DCI-based implementation of TM consists of two roles. The first is the source account and the second is the sink account. The assumption is that the user has a number of accounts such as a savings account, investment account, family account and the other type of accounts. These types of accounts can select to play either of the two roles.

As can be seen from the code below, each role is defined using a JavaScript class. It is worth noting that both the source and sink accounts are methodful roles. A methodful role is a role with some role-specific methods. The Money sink role should have a method to perform depositing money operation. The method deposit(int) is designed for that purpose. The sink account role is implemented by the class structure and is presented below.

```
function TransferMoneySink()
{
     this.depositAmount = function( amount )
     {
          this.deposit( amount );
          this.updateLog( "Transfer in", new Date(), amount );
     };
};
```

The method withdraw(int) is designed for Money Source role to perform withdrawing money operation. Money Source role implemented through class is presented below.

```
function TransferMoneySource()
{
       this.transferTo = function( amount )
       {
            if ( this.availableBalance() < amount )
           {

                   log.error(this.toString() +
                   ": Insufficient funds!");
                   return;
            }
            else
            {
                  this.withdraw( amount );
                  this.updateLog( "Transfer Out", new Date(),
                  amount );
                  this.context.sink.depositAmount( amount );
            }
        };
   }
```

In the codes presented above, both of the roles have an updateLog() method for logging purposes. After withdrawing, the Money Source role calls depositAmount() method of the Money Source role. This calling is handled through the Context object.

3.1.1.3 Context

Context is responsible for mapping the roles to the data objects. Below, two versions of the Context implemented are shown. As can be seen Context class consist of the roles and data objects selected to be map to those roles.

The two versions have one slight difference. The first version directly maps the roles to the data objects. But, the second version makes a separate object consisting the roles and data objects and then it defines the doit(int) method to be used by the program to trigger money transfer Context. The difference will cause a different style of using Context in the programme. The explained difference is also visible in the following codes. The first version of the Context is presented below.

```
function TransferMoneyContext( accounts )
{
    Roles.Context( this,
    {
           "source" :
           {
               "object" : accounts["from"],
               "roles" : [ TransferMoneySource ]
           },
           "sink" :
           {
                "object" : accounts["to"],
                "roles" : [ TransferMoneySink ]
           }
    } );

    var context = this;
    return function( amount )
    {
           context.source.transferTo( amount );
    };
}
```

The second version of the Context is presented below.

```
function TransferMoneyContext2( accounts )
{
    var context = {};
    Roles.Context( context,
    {
        "source" :
        {
             "object" : accounts["from"],
             "roles" : [ TransferMoneySource ]
        },


        "sink" :
```

```
        {
             "object" : accounts["to"],
             "roles" : [ TransferMoneySink ]
        }
    } );

    context.doit = function( amount )
    {
         context.source.transferTo( amount );
    };
    return context;
}
```

From the code above, the mapping duty of the context is illustrated. The context is mapping the “from” and “to” data objects to the “TransferMoneySource” and “TransferMoneySink” roles previously defined.

3.1.1.4 Executing the Context

Below, two examples illustrate the use of context to run the MT operation. First, two account objects with an initial amount of money are instantiated from account class. Hence, the TM context is created and the source (“from”) and sink (“to”) accounts are mapped to mySavingsAccount and myInvestmentAccount. Then by calling the constructor of the context, the TM context is triggered.

Example 1

```
function example1()
{
           var mySavingsAccount = new Account( "Savings",
           10000 );
           var myInvestmentAccount = new Account( "Invest-
           ment", 0 );
           var transfer = new TransferMoneyContext( {
                      "from" : mySavingsAccount,
                      "to" : myInvestmentAccount
           } );
           transfer( 100 );
}
```

Example 2

```
function example2()
{
           var mySavingsAccount = new Account( "Savings",
           10000 );
           var myInvestmentAccount = new Account( "Invest-
           ment", 0 );

           var transferContext = new TransferMoneyContext2(
           {
                      "from" : mySavingsAccount,
                      "to" : myInvestmentAccount
           } );
           transferContext.doit( 100 );
}
```

The second example runs the second version of context previously defined. This is considered as the only difference between the two examples. Because of using context the triggering will be done by calling doit(int) method through the transferContext object.

3.1.2. C# Implementation

I have studied Christian Horsdal's Implementation (Horsdal 2009) of the TM, which was implemented using C# programming language. The Account class representing the Data is an abstract class and contains the basic operation which includes Deposit() method for depositing money and Withdraw() method for withdrawing.

3.1.2.1 Data

```
Public abstract class Account
{
            Public double Balance
            {
                        get; protected set;
            }

            public void Withdraw(double amount)
            {
                        Balance -= amount;
            }

            public void Deposit(double amount)
            {
                        Balance += amount;
            }
            ...
  }
```

As already has been explained earlier the Account class is an abstract class. Hence, the instantiation of this class is not possible. Therefore, in order to have several account objects, some other classes such as SavingsAccount class that is not an abstract class and inheriting from the base account class is necessary.

```
public class SavingsAccount: Account, TransferMoneySource,
TransferMoneySink
{
            public SavingsAccount()
            {
                        Balance = 1000;
            }
            ...

}
```

The code is showing the SavingsAccount class that inherits the base Account class it's putting some initial value for the Balance property. The TransferMoneySource and TransferMoneySink are C# interfaces for implementing the Interaction that will be discussed in the following section.

3.1.2.2 Interaction

In the Interaction the roles and their functionality are defined. The Interaction roles are defined using the C# interface structure. Below is the code representing TM Source role implementation using C# interface structure.

```
Public interface TransferMoneySource
{
          double Balance { get; }
          void Withdraw(double amount);
          void Log(string message);
}
```

Below is the code representing TM Sink role implementation using C# interface structure.

```
Public interface TransferMoneySink
{
          void Deposit(double amount);
          void Log(string message);
}
```

As illustrated above, the roles are implemented with C# interfaces. Although, the roles should maintain their functionalities, for this reason C# extension method is used.

Extension methods allow the researcher to "add methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type. For client code written in C# and Visual Basic, there is no apparent difference between calling an extension method and the methods that are actually defined in a type." (Microsoft MSDN Website 2013).

The TM trait implemented with extension method injects the TransferTo() method to the TM source role. In other words, this injection is to all objects of saving accounts, which plays the TM source role in the Context. The TransferTo() method will first check the source money account for availability of enough fund to be transferred. When the transfer happens, the method will withdraw the amount of money and deposit it to the account which is playing the Sink role in the Context.

```
public static class TransferMoneySourceTrait
{
          Public static void TransferTo(
          This TransferMoneySource self,
          TransferMoneySink recipient, double amount)
          {
             // The implementation of the use case
             If  (self.Balance < amount)
             {
                 Throw new ApplicationException(
                 "insufficient funds");
             }
```

```
                    self.Withdraw(amount);
                    self.Log("Withdrawing " + amount);
                    recipient.Deposit(amount);
                    recipient.Log("Depositing " + amount);
                }
}
```

3.1.2.3 Context

The Context should map the TM source and TM sink roles to two SavingAccount objects and then run TransferTo() method of the TM source by Execute() method. The constructor of the context class has two parameters of the type TransferMoneySource and TransferMoneySink in its parameter list. SavingAccount class is implementing TransferMoneySource and TransferMoneySink interfaces. Any object of SavingAccount class could be sent to the context while making an object of context class. The duty of Execute() method is to run the context in order to perform the money transferring operation between the source and sink accounts. The following code represents the TransferMoneyContext class.

```
public class TransferMoneyContext
{
        Public TransferMoneySource Source
        {
                get; private set;
        }

        Public TransferMoneySink Sink
        {
                get; private set;
        }

        Public double Amount
        {
                get; private set;
        }

        public TransferMoneyContext(TransferMoneySource
        source, TransferMoneySink sink, double amount)
        {
                Source = source;
                Sink = sink;
                Amount = amount;
        }

        public void Execute()
        {
                Source.TransferTo(Sink, Amount);
        }
    }
```

3.1.2.4 Executing the Context

The following code illustrates how to execute the context to perform the transfer of money. The first two lines of code define two saving account objects. Then an object of

TransferMoneyContext is created. The defined saving account objects as the Source and Sink accounts plus amount of money to be transferred is sent as an argument to the constructor. Then the Execute() method is called to trigger the context to do the money transfer operation.

```
SavingsAccount src = new SavingsAccount();
SavingsAccount snk = new SavingsAccount();
new TransferMoneyContext(src, snk, 500).Execute();
```

3.2. DCI Implementation – Messaging Example

The purpose of this implementation is; fist to implement a small example based on DCI and then try feature adding and removal in practice. The messaging example is a system with several users in which each user is able to communicate with the other users via messaging. Every user has an Inbox for the received messages and an Outbox for the sent messages. We want to know how a simple message transfer operation can be done using DCI architecture.

3.2.1. C# Implementation

3.2.1.1 Data

The Data is a User class. Each object of the User class represents a user who can interact with messaging system. Each object of the User class has two arrays; the Inbox for storing the received message and Outbox for storing sent messages. Two methods of saveIn() and saveOut() store the received messages to Inbox array and sent messages to Outbox array respectively.

The following code illustrates the main points of the User class. The Interaction in the C# version of DCI is implemented using C# interface structure as has been noted in the TM example. MessageSendingReceiver and MessageSendingSender are the interfaces for the possible roles that are defined in the following section.

```
public class User : MessageSendingReceiver,
MessageSendingSender
{

        ...
        public string[] inbox = new string[MAILBOX_SIZE];
        public string[] outbox = new string[MAIL-
        BOX_SIZE];

        public string Name
        {
                get; set;
        }
```

```
                public void saveIn(string message)
                {
                        ...
                }

                public void saveOut(string message)
                {
                        ...
                }
                        ...
}
```

3.2.1.2 Interaction

Interaction includes two roles. The first role is called Sender and the second is called Receiver. Several users can be defined and each user can play the role of Receiver or Sender in one context. The Sender role; a C# interface, has saveOut() method for saving the sent messages in the Outbox array of Sender object. Receiver role has a method named saveIn() for saving the sent messages by the sender in the Inbox array of receiver object. Each Sender is supposed to have send() method. The duty of the send() method is to run the saveOut() method of the Sender object and saveIn() method of the Receiver object. The send() method is implemented with the C# extension method which explained in previous example.

```
public interface MessageSendingSender
{
            string Name
            {
                        get; set;
            }
            void saveOut(string message);
}

public interface MessageSendingReceiver
{
            string Name
            {
                        get; set;
            }
            void saveIn(string message);
 }

public static class MessageSendingSenderTrait
 {
            public static void send(this MessageSendingSender
            sender,MessageSendingReceiver reciver, string
            message)
            {
                        sender.saveOut("||" + reciver.Name +
                        "||→  " + message);
                        reciver.saveIn("||" + sender.Name +
                        "||←  " + message);
            }
    }
```

3.2.1.3 Context

Context maps the Sender and Receiver roles to two User objects and run the send() method of Sender object using DoIt() method. The constructor of the Context class (MessageSendingContext) has Sender and Receiver objects and message text parameters. Since the user class implements (inherits) both MessageSendingReceiver and MessageSendingSender interfaces, we can easily pass the User objects instead.

```
class MessageSendingContext
{
            public MessageSendingSender Sender
            {
                        get; private set;
            }
            public MessageSendingReceiver Receiver
            {
                        get; private set;
            }
            public string Message
            {
                        get; private set;
            }
            public MessageSendingContext(MessageSendingSender
            sender,MessageSendingReceiver receiver, string
            message)
            {
                        Sender = sender;
                        Receiver = receiver;
                        Message = message;
            }
            Public void DoIt()
            {
                        Sender.send(Receiver, Message);
            }
}
```

3.2.1.4 Executing the Context

In order to execute Context, an instance of the context class is created. The constructor of the Context has too User objects in its parameter list. Thus, two User objects are passed as arguments to constructor of the Context. If DoIt() method of context object is called, the sending process will be started. In this example the execution takes place when the user of the application selects two users among all users through the GUI (Figure 3.1) and types the message in a related textbox and then presses the send button. Here is the code related to the click event of send button.

```
private void btnSend_Click(object sender, EventArgs e)
{
            if (cboTo.SelectedIndex >= 0 &&
            cboFrom.SelectedIndex >= 0 &&txtMessage.Text!="")
            {
```

```
                                    new MessageSendingCon-
                                    text((User)cboFrom.SelectedItem,
                                    (User)cboTo.SelectedItem, txtMes-
                                    sage.Text).DoIt();
                                    ...
                }
        }
```

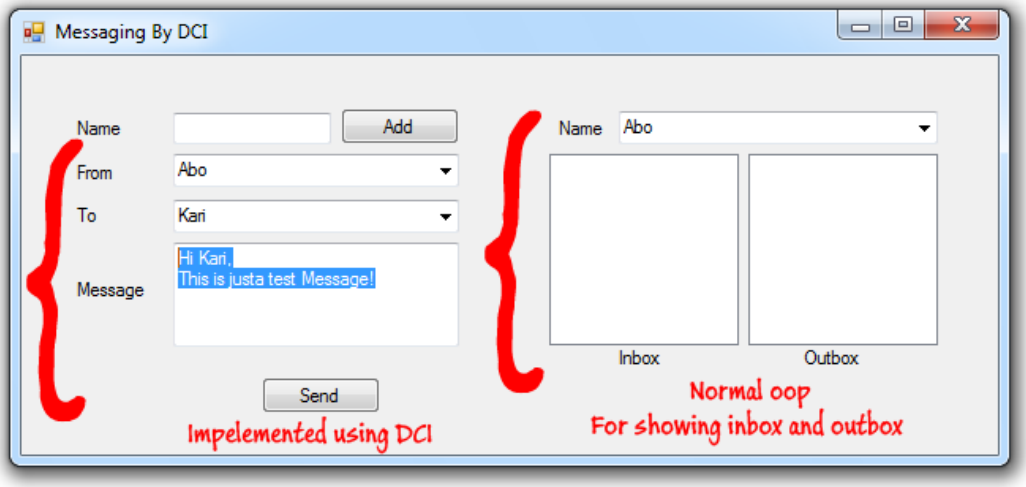

Figure 3.1. *Messaging example application UI – C# Implementation*

3.2.2. JavaScript Implementation

The messaging example is also implemented with JavaScript using the DCI framework that was designed by Anders Nawroth (Nawroth 2009). He designed the DCI framework as a separate unit that can be added as a separate library to projects that aim to use the DCI architectural style.

3.2.2.1 Data

In the messaging example, Data consists of the User class, which represents the real user. Each user is one object of this class. Each User object includes two arrays for storing the received and sent messages. The Outbox and Inbox arrays are defined for this purpose. Two methods, saveIn() and saveOut() store the received messages to inbox array and store messages that have been sent to the outbox array. The printInbox() and printOutbox() methods show the Inbox and Outbox content into a HTML-based user interface.

```
function User( userName)
{

            var inbox = new Array();
            var outbox = new Array();
            var name = userName ;
```

```
            this.saveIn = function( message)
            {
                        inbox.push(message);
            };

            this.saveOut= function( message)
            {
                        outbox.push(message);
            };
            ...

            this.printOutbox = function()
            {
            ...
            };

            this.printOutbox = function()
            {
                        ...
            };
}
```

3.2.2.2 Interaction

In the Messaging example, Interaction includes two roles. The first role is called the Sender and the second is called the Receiver. As can be seen in the section above, we have several users in which each user can play the role of Receiver or Sender in specific context. Each role has a class in which all Role Methods are defined.
The Sender role has saveMessageOut() method for saving the sent messages in its outbox array by running saveOut() method. The Receiver role has saveMessageIn() method for saving the sent messages by the sender in its inbox array. Saving the sent messages is done by the SaveIn() method.

```
function MessageSendingSender()
{
            this.Name = function ()
            {
                        return this.getName();
            };

            this.saveMessageOut = function( message )
            {
                        this.saveOut( message );
            };
}

function MessageSendingReceiver()
{
            this.Name = function ()
            {
                        return this.getName();
            };
            this.saveMessageIn = function(message )
            {
                        this.saveIn( message );
            };
}
```

3.2.2.3 Context

In the Context, *Data* objects should be mapped to their related roles in the Interaction. Context runs saveMessageOut() method of Sender role for saving the messages inside its inbox and runs saveMessageIn() method of Receiver role for saving sent messages inside the receiver's inbox.

```
Function MessageSendingContext( users )
{
            Roles.Context(this,
            {
                        "sender" :
                        {
                          "object" : users["from"],
                          "roles" :  [ MessageSendingSender]
                        },

                        "receiver" :
                        {
                          "object" : users["to"],
                          "roles" :  [ MessageSendingReceiver]
                        }
            } );

            var context = this;
            return function( message )
            {
                        context.sender.saveMessageOut("["+
                        context.receiver .Name()+ "]: "+ mes-
                        sage );
                        context.receiver.saveMessageIn("["+
                        context.sender.Name()+ "]: "+ message)

            };
}
```

3.2.2.4 Executing the Context

Two User objects to play the Sender and Receiver roles are needed. The messageSendingContext should be called in order to send the two created User objects; the Sender and Receiver as the arguments. Then to perform the message sending operation the context should be run.

```
var abo = new User( "Abolfazl");
var kari = new User( "Kari" );
var msgSend = new MessageSendingContext( {
                                          "from" : abo,
                                          "to" : kari,
                                        } );

msgSend(" test message text ");
```

To run and test the above JavaScript implementation, test program (messagesending-test.js) with a couple of User class objects is defined. Besides, some usage example is

also implemented. Moreover, the HTML interface for interacting with the messaging example is also implemented as shown in figure 3.2.

Kari Systä		Abolfazl Hoseini		BigBrother	
Inbox	Outbox	Inbox	Outbox	Inbox	Outbox
[Abolfazl]: Hi	[Abolfazl]: Hello!	[Kari]: Hello!	[Kari]: Hi	[Abolfazl to Kari] Hi [Kari to Abolfazl] Hello!	

Figure 3.2. *Messaging example application HTML UI – JavaScript Implementation*

4. RESULTS AND FINDINGS

4.1. From the DCI Paradigm to Feature-Oriented Software

Software that is developed with the DCI architecture has one or more contexts. Roles are involved in those contexts in which each context is responsible to implement certain use case scenarios. Each context represents a feature or a group of features. A feature is a unit of functionality with a value. Features are designed for the customers and each feature is independent from the point of applicability.

Messaging example is implemented in section 3. This use case has a context called messageSendingContext and the Sender and Receiver as the context roles. The messaging example feature is developed by the Context and the Interaction between the roles (Sender and Receiver).

This section aims to add a new feature named Logging Feature. This feature is responsible for recording some logs in the sending process and it is added to the existing message sending context. Unlike the message-sending feature, the logging feature does not construct a new context and it is added to the existing messaging example context. The next section describes this feature.

4.2. Logging Feature

By having messaging example implemented with DCI architectural style, now I practice feature adding process. Logging component is a new feature added to Messaging example. It is responsible for sending a copy of each transferred/sent message to a specific user as a log. For the specific user to receive the log, it is necessary to play a new role, which is named BigBrother.

BigBrother is simply assumed as a user in the Messaging example. BigBrother itself can send and receive messages like all other users. The only difference of BigBrother with an ordinary user is that when the sender sends a message to the receiver, a copy of the message will be sent to BigBrother as a log. The logging component is considered a feature and it is added to the messaging example. The BigBrother role is included in the Logging feature. The difference between the BigBrother role and Logging component should not be confused. BigBrother is a role and the Logging component is a feature.

Below are some changes that should be made in the existing messaging example to include the logging feature. The first change that must be applied to existing application is to add new role to Interaction. The BigBrother role with its properties and functionalities are defined. This role has a method named logIt() to copy every single message that passes between users to its inbox.

4.2.1. Defining BigBrother's role

4.2.1.1 C# Implementation

BigBrother's Interface is created and added to the existing code.

```
public interface MessageSendingBigBrother
{
            string Name
            {
                        get; set;
            }

}
```

Note: This newly added interface has to be implemented or inherited by the User class. The change can be seen in the class definition. The highlighted part is what is added:

```
public class User: MessageSendingReceiver,
MessageSendingSender, MessageSendingBigBrother
{

            //the User class body is untouched.
}
```

BigBrother's trait for logIt() function is added. In this implementation, the trait is a static class that includes some static methods. These methods will be injected to the BigBrother interface throughout the C# extension method. In terms of DCI, this trait will add the BigBrother's functionalities to the RolePlayer object.

```
public static class MessageSendingBigBrotherTrait
    {

            public static void logIt(this MessageSending-
            BigBrother brother, string message)
            {
                        ((User)brother).saveIn(message);
            }
   }
```

In the code above, the 'brother' parameter is an object with a BigBrother role. For accessing the saveIn() method the 'brother' parameter should downcast to the User class type. This down-casting is against the concept of DCI according to the researcher's idea. The researcher will extend this argument in the Discussion section.

4.2.1.2 JavaScript Implementation

BigBrother's role is created using a class in JavaScript.

```
Function MessageSendingBigBrother()
{
            this.logIt = function(message)
            {
                        this.saveIn(message);
            };
}
```

4.2.2. Modifying the Context

The second change should be made in the Context. The newly created role should be added to the Context. Besides, the *logIt()* method of BigBrother's role should be triggered in the Context. The new lines of added codes are highlighted below.

4.2.2.1 Modifying C# version

In the following, the modified version of C# code for Context is shown.

```
Class MessageSendingContext
{
            Public MessageSendingSender Sender
            {
                        get; private set;
            }
            Public MessageSendingReceiver Receiver
            {
                        get; private set;
            }
            Public MessageSendingBigBrother BigBrother
            {
                        get; private set;
            }

            public string Message
            {
                        get; private set;
            }

            public MessageSendingContext(MessageSendingSender
            sender,MessageSendingReceiver receiver, Messag-
            eSendingBigBrother bigBrother, string message)
            {
                        Sender = sender;
                        Receiver = receiver;
                        BigBrother = bigBrother;
                        Message = message;
            }
            public void DoIt()
            {
                        Sender.send(Receiver,BigBrother, Mes-
                        sage);
                        BigBrother.logIt(Sender.Name, Re-
                        ceiver.Name, Message);
      }
   }
```

4.2.2.2 Modifying JavaScript version

In the following, the modified version of JavaScript code for Context is shown.

```
function MessageSendingContext( users )
{

    Roles.Context(this,
    {
        "sender" :
        {
            "object" : users["from"],
            "roles" :  [ MessageSendingSender]
        },
        "receiver" : {
            "object" : users["to"],
            "roles" :  [ MessageSendingReceiver]
        },
        "bigBrother" : {
            "object" : users["bigBrother"],
            "roles" :  [ MessageSendingBigBrother]
        }
    } );

    var context = this;
    return function( message )
    {
        context.sender.saveMessageOut("["+  context.receiver  .Name()+
        "]: "+ message );
        context.receiver.saveMessageIn("["+ context.sender.Name()+ "]:
        "+ message );
        context.bigBrother.logIt("["+context.sender.Name()+" to "+ con-
        text.receiver.Name()+ "] "+  message);
    };
}
```

4.3. Bug Found in JavaScript Implementation

A bug in the DCI implementation in JavaScript by Anders Nawroth was found. The problem is that The DCI framework designed by him could not handle methodful roles with more than one method. However, Prof. Kari Systä and Mr. Jari-Pekka Voutilainen have solved the bug and the corrected version of DCI is used for the implementation of the examples.

5. DISCUSSION

5.1. Differences in JavaScript and C# implementation

DCI is a paradigm consisting of three different parts: Data, Context and Interaction. DCI's different parts carry certain duties and responsibilities. But when applying the DCI concept to a particular programming language, the limitation of the language causes some problems for DCI architecture. This limitation results in different structures for different languages.

DCI has been implemented in different languages from which JavaScript and C# are selected for this thesis. Their differences in regard to DCI implementation are presented in section 3. In this section, the researcher aims to present how DCI Implementation is influenced by language-specific characteristics.

The Data part of DCI in both JavaScript and C# is implemented using class structure, although implementation of Interaction part in JavaScript and C# slightly varies. In JavaScript each role is a class and the Data class will inherit from the role class. But, C# does not support the multiple inheritance and roles cannot be constructed with class structure. Hence, the Interface structure should be used. The fact that each class can inherit or implement multiple interfaces allows having several roles with several interfaces. The interface method has no body. What is needed to be included in the roles' interface is the header or declaration part of the methods. And programmer need to make existing class to implement new interfaces.

There are two possibilities to define the body of the declared methods in the interface: The first is to have the definitions in the Data class that implements those interfaces. However, this possibility can lead to some problems. The first problem is that, role methods should be defined in the Data class and Data class holds all the methods of the roles. Therefore, the size of the Data class is expanded. The next problem is actually sharing the methods of all roles in the Data class, and this does not sound technically appropriate. I believe the Data should contain the general functionality of all objects regardless of the roles they may play in the Context.

The second approach is to use the extension method mechanism to inject the methods to the roles. The injected method is called trait. This makes it possible to have the methods in a separate location than the interface and then inject those methods to the interface and then call them with the object that inherits those interfaces. This approach may raise some issues such as no accessibility of roles to the data object members inside the trait. In order to access the Data object members, the object playing the role should be down-casted to the original type (Data object). Down-casting is against the concept of encapsulation. By down-casting to the original Data object, the object gets the access to all of the Data class members in which it should only access its own (role) methods.

The Context part of DCI is similar in both JavaScript and C#. The Context is constructed using a class structure. The context class in JavaScript is more efficient and readable because it uses JSON-alike syntax to map the data objects to the roles. This is possible by a framework designed by Anders Nawroth.

The last discovered difference of implementing DCI with JavaScript and C# is that there is a framework provided for application of DCI to assist JavaScript implementation while in C#, such framework has not yet been provided. This DCI framework assists the Context to map the roles to the Data objects.

5.2. DCI Concept and Code Structure

DCI tries to capture different use case scenarios. In DCI, each use case is connected to a Context. Context maps roles in the Interaction with some objects in Data. The researcher believes – according to DCI studies – that the DCI tries to say that each use case is connected with a Context.

The researcher found it puzzling that in DCI literature the actual functionality is implemented in one of the roles and it is triggered by the Context. For example in MT example, the use case is about transferring money. However, the actual money transferring is not done by the Context but by a sender. In fact, the sender has the transferTo() method which takes care of the actual transferring. The doit() method of Context calls the transferTo() method. The researcher believes that this transferTo() method is neither for the Source nor for the Sink roles but it should belong to the Context.

If one role is responsible for performing the transfer should it be the Source role or the Sink role? If it were the Source role, then why not opt for the Sink role and vice versa? It is logical that Source account performs actual transformation. But, someone may argue and try to justify this. Considering defining the shortest path between two nodes in a graph, one node plays the role of the starting point and the next node plays the ending point. But, from what point; starting or ending; should the process begin? Should it be started from the starting point, ending point, or the Context? It is clear that finding the shortest path between the two points is out of the capabilities of the two roles. Hence, the Context is responsible for the process.

Moreover, it is beneficial to declare and assign the responsibilities to the Context. This makes it possible for the Context to be inherited by the other Context. Certain functionalities can be used by other Context too. Besides, additional functionalities can be added to the new Context as well. This makes the hierarchy of inherited Context possible.

5.3. DCI Challenges Faced

Studying DCI and DCI implementation in JavaScript and C# has challenged the researcher. The challenges are the following:

For a better implementation of DCI in regard to the specific DCI characteristics, DCI needs to be very well supported by different programming languages. Supports are the

method injection to the roles and mapping the roles to the existing Data objects. Programming language differences have a direct effect on the implementation of DCI and the result quality. Some programming languages cause limitations for DCI implementation. Such limitations of C# and JavaScript implementations are presented in section 3.

The next limitation is that DCI architecture is not applicable to implementation of all applications and use cases. For some application ideas, it is not applicable to use DCI architecture for the application development. However, DCI can be an applicable paradigm for many other application ideas because of the separation of Data, Context, and Interaction and defining the roles and the possible scenario is possible.

To implement DCI in both JavaScript and C#, the researcher has looked for appropriate use cases. In this process, the researcher has encountered the two following issues: Despite the use cases that can be implemented using DCI, some use cases were found but DCI architecture is not applicable for implementation. And, some use cases were found that can use DCI but DCI does not have added values compare to OOP. I believe that in these scenarios the state of the Data objects will not change while Context executes. For example, to implement a scenario to find the shortest path between two nodes in a graph, the Data, Context and Interaction are defined and separated. The Data part is the Node class in which each object represents a node in a graph. The Interaction consists of the starting node role and destination node role. The process is to find the shortest path between the starting and destination node.
This process raises one question; is the state of the Data objects that are playing the starting and destination roles changed while the process of finding the shortest path is running? The possible answer is No. Hence, if the state of the Data objects is not changed and the only target is to find the shortest path why not use OOP and the static method in a utility class as following:

```
Utility.findShortestPath(NodeA, NodeB);
```

I believe that using OOP for such use cases is less complex.

5.4. Feature Removal Expectation

The target to construct feature-oriented software is explained earlier (section 2). However, it is a good decision to do extra work which the use of DCI requires. One general benefit of feature-oriented software is that the focus is around the features on the project. Dealing with meaningful units – features – is much easier than dealing with lines of code.

In addition, the next essential benefit is having proper feature management for the project. Feature management is not only about adding some new features but also removing unnecessary or deprecated features. Removing the unnecessary features adds value to the whole project. Besides, removing the unnecessary features has a positive impact on reducing the cost of development of a project and also the cost of maintenance. Moreover, removing unnecessary features prevents the project from resulting in a bloated software

or bloatware. Bloatware is software with many unnecessary features. These kinds of system have some consequences such as unnecessary memory allocation and network usage. In most cases, bloatware causes speed and space problems for the software.

While we are using the DCI architectural style for achieving a feature-oriented programme, Data is mostly untouched and we are working with the Context and Interaction to define new roles and bind them to the data objects. For removing a feature the following steps are taken into an account:

- First, all the roles related to the feature should be removed. Roles can be removed after the two following steps:
    - First, removing the methods being injected to the roles.
    - Last, removing the roles class themselves.
- Last, some changes should be made in the *Context*, which represents a network of roles. Changes can be made after the two following steps:
    - First, the roles from the introducing part or where the mapping roles to data objects occur should be removed.
    - Last, the last change is to the trigger function of Context called DoIt(). All the codes relating and connecting the feature should be removed from the DoIt() method.

5.5. Feature Dependencies and Feature Removal

The importance of feature removal and the possibility to remove the features that are constructed with DCI was explained. However, one problem arises and that is the problem of feature dependencies. Some features may be connected to each other and have some dependencies. Feature dependencies could be so that the features are indigenously dependent to each other or the dependency is accidental and resulted by implementation. Removing one feature may result in a problematic situation for the dependent feature.

One solution is that all features should be independent enough, so that when one feature is removed, it does not affect the other features. The other solution is that if the programming language is dynamic enough, dependency is possible and feature removal is possible. However, the existence of one feature should be checked first. If it exists it should be used in the context, but if not then an alternative code should be run. It is worth mentioning that some features are congenitally dependent to each other. The dependency between them should be considered separately from the dependency in their implementation.

5.6. Combination of DCI and AOP Resulting Feature-Oriented Software

I believe that the combination of AOP and DCI can be an effective change. A feature-based system can be constructed using DCI and AOP in such a way that aspect is connected with several roles in the Interaction in order to run the process. However, I believe that such combination theory requires further research.

6. CONCLUSION

By the study of relevant literature and implementation, I conclude that DCI can be used in feature-oriented software development. DCI is a very interesting and valuable paradigm and software architecture as some authors claim. DCI targets the user's mental model and connects the user's mental model to the system. Moreover, DCI increases the readability and maintainability of the code.

I further conclude that DCI is not applicable for all programming languages and it is not a suitable paradigm for all the possible use case scenarios. Therefore, it is important for programmers – if DCI is the selected paradigm – to bear in mind the DCI limitations discussed earlier in this thesis, in order to select the appropriate programming language for the development. Furthermore, the characteristics of the scenarios are also vital when DCI is the selected paradigm for the development since not all scenarios work well with DCI.

I also conclude that in order to be able to perform feature removal practice, software development has to be based on its features. DCI characteristics helps to result in such feature-oriented software.

REFERENCES

Apel, S. and Kästner, C. (2009). An Overview of Feature-Oriented Software Development. *J. Object Technology (JOT)*, Vol. 8, No. 5, pp. 49–84.

Batory, D. (2005). Feature Models, Grammars, and Propositional Formulas. *International Software Product Line Conference (SPLC)*. Vol. 3714, pp. 7–20. Springer-Verlag.

Baxter, I. and Mehlich, M. (2001). Preprocessor conditional removal by simple partial evaluation. *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, pp 281–290. IEEE Computer Society, Washington, DC, USA.

Braak, T. (2006). Extending Program Slicing in Aspect-Oriented Programming with Inter-Type Declarations. *5th Twente student conference on IT*. University of Twente, Enschede, Netherlands.

Bruntink, M., van Deursen, A., D'Hondt, M. and Tourwe´, T. (2007). Simple Crosscutting Concerns Are Not So Simple: Analysing Variability in Large-Scale Idioms-Based Implementations. *Proceedings of the 6th international conference on Aspect-oriented software development (AOSD '07)*, pp. 199–211. ACM, New York, NY, USA.

Campbell, L.J., Halpin, T.A. and Proper, H.A. (1996). Conceptual Schemas with Abstractions – Making flat conceptual schemas more comprehensible. *J. Proper, Data & Knowledge Engineering*, Vol. 20, No. 1, pp. 39–85.

Cataldo, M. and Nambiar, S. (2012). The Impact of Geographic Distribution and the Nature of Technical Couplingon the Quality of Global Software Development Projects. Forthcoming. *J. Software evaluation and process*, Vol. 24, No. 2, pp. 153–168.

Coady, Y. and Kiczales, G. (2003). Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. *Proceeding Second Int'l Conference Aspect-Oriented Software Development*, pp. 50–59.

Coutaz, J. (2006). Meta-user interfaces for ambient spaces. *Proceedings of the 5th international conference on Task models and diagrams for users interface design (TAMODIA'06)*, pp. 1–15. Springer-Verlag, Berlin, Heidelberg.

Coplien, J.O., (2010). Restoring function and form to patterns: DCI and agile expression of architecture. *Software Architect,* London.

Coplien, J.O., (2012). Objects of the people, by the people, and for the people. *11th annual international conference on Aspect-oriented Software Development Companion*. New York, NY, USA.

Coplien, J. O. and Bjørnvig, G. (2010). Lean Architecture for Software Development. ISBN 978-0-470-68420-7, Wiley, Chichester, United Kingdom.

Curteanu, M. (2010). Using the Model-View-Controller for Creating Applications for Project Management. *J. Open Source Science,* Vol. 2, No. 4.

Curtis, B. (1986). Human Factors in Software Development. *IEEE Computer Society*.

Emonet, R. (2011). Environment – Application – Adaptation: a Community Architecture for Ambient Intelligence. *The First International Conference on Ambient Computing, Applications, Services and Technologies.*

Gulzar, N. (2002). Fast Track to Struts: What it Does and How, available at http://media.techtarget.com/tss/static/articles/content/StrutsFastTrack/StrutsFastTrack.pdf, last visited Jan. 1 2014.

Günther, S. and Sunkle, S. (2012). rbFeatures: Feature-oriented programming with ruby. J. *Science of Computer Programming*, Vol. 77, No. 3, pp. 152–173.

Hannemann, J. and Kiczales, G. (2002). Design Pattern Implementation in Java and AspectJ. *Proceeding 17th ACM Conference Object-Oriented Programming, Systems, Languages, and Applications*, pp. 161–173.

Hayata, T., Jianchao Han, Beheshti, M. (2012). Facilitating Agile Software Development with Lean Architecture in the DCI Paradigm, *Information Technology: New Generations (ITNG) 9th International Conference,* pp. 343–348.

Horsdal, C. (2009). Implementation of DCI in C-Sharp, Source code available at http://www.horsdal-consult.dk/2009/05/dci-in-c.html, last visited Jan. 25 2014.

Jacobson, I. and Ng, P.W. (2005). Aspect-Oriented Software Development with Use Cases, ISBN 978-0-321-26888-4, Addison-Wesley Professional.

Kavand, M., Paarsa, S., Faraahi, A. (2011). ciFeature: A context-independent feature-oriented software development approach, *Computer Science & Education (ICCSE) 6th International Conference.*

Krasner E. G. and Pope S.T. (1988). A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. J. Object Oriented Programming, Vol. 1, No. 3, pp. 26–49.

Kästner, C., Apel, S., ur Rahman, S., Rosenmüller, M., Batory, m. and Saake, G. (2009). On the Impact of the Optional Feature Problem: Analysis and Case Studies. *International Software Product Line Conference (SPLC).* SEI, pp. 181–190.

Kästner, C., Apel, S. (2013), Generative and Transformational Techniques in Software Engineering IV Lecture Notes in Computer Science, J. *Feature-Oriented Software Development*, ISSN 0302-9743, Springer Berlin Heidelberg, pp. 346–382.

Key Miller, S. (2001). Aspect-Oriented Programming Takes Aim at Software Complexity. *J. Computer*, Vol. 34, No. 4, pp. 18–21.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. (2001). An overview of AspectJ, *Proceeding ECOOP '01 Proceedings of the 15th European Conference on Object-Oriented Programming*, pp. 327–354.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M. and Irwin, J. (1997). Aspect-Oriented Programming. *11th European Conference of Object-Oriented Programming*, pp. 220–242.

Kutschera A. (2011). *A comparison of DCI and SOA in Java*, available at http://www.maxant.co.uk/download.jsp?uid=11, last visited Jan. 1 2014.

Laddad, R. (2003). AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co., Greenwich, CT, USA.

Lieberherr, K., Holland, I. and Riel, A. (1988). Object-oriented programming: an objective sense of style. *Proceedings Conference on Object-oriented programming systems, languages and applications (OOPSLA '88),* Vol. 23, No. 11, pp. 323–334. Norman Meyrowitz (Ed.). ACM, New York, NY, USA.

Magdaleno, A. M., Cláudia Maria Lima Werner, Renata Mendes de Araujo, Reconciling software development models: A quasi-systematic review, *J. Systems and Software*, Vol. 85, No. 2, pp. 351-369.

Madeyski and Szała. (2007). Impact of aspect-oriented programming on software development efficiency and design quality. *Software, IET*, Vol. 1, No. 5, pp. 180–187.

Microsoft MSDN. (2013). Extension methods, available at http://msdn.microsoft.com/en-us/library/bb383977.aspx, last visited Jan. 1 2014.

Murphy, G. (2006). Aspect-Oriented Programming. *J. IEEE Software*. Vol. 23, No. 1, pp. 20–23.

Murphy, G., Walker, R. and Baniassad, E. (1999). Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-Oriented Programming. *J. IEEE Transactions on Software Engineering*, Vol. 25, No. 4, pp 438–455.

Nagappan, N. and Ball, T. (2007). Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. *1st Int'l Symposium on Empirical Soft. Eng. and Measurement* (Madrid, Spain). *ESEM'07*.

Nawroth, A. (2009). Implementation of DCI in JavaScript, Source code available at http://svn.nornix.org/dci/trunk/WebContent/index.html, last visited Jan. 1 2014.

Parnas. D. L. (1976). On the design and development of program families. *Software Engineering, IEEE Transactions on (TSE)*, Vol. SE-2, No. 1, pp. 1–9.

Platunov, A. and Nickolaenkov, A. (2012). Aspects in the Design of Software-Intensive Systems. *Mediterranean Conference on Embedded Computing.* Bar, Montenegro.

Qing, C. and Zhong, Y. (2012). A Seamless Software Development Approach Using DCI. *IEEE 3rd International Conference on Software Engineering and Service Science (ICSESS)*, pp. 139–142, Beijing.

Rashid, A., Cottenier, T., Greenwood, P., Chitchyan, R., Meunier, R., Coelho, R., Südholt, M. and Joosen, W. (2010). Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe. *J. Computer*, Vol. 43, No. 2, pp. 19–26.

Reenskaug, T. (2003). The Model-View-Controller (MVC). *Its Past and Present*. Department of Informatics available at http://heim.ifi.uio.no/trygver/2003/javazone-jaoo/MVC_pattern.pdf ,last visited Jan. 1 2014.

Reenskaug, T. (2008). The Common Sense of Object-Oriented Programming. Department of Informatics, available at http://folk.uio.no/trygver/2008/commonsense.pdf, last visited Jan. 1 2014.

Reenskaug, T. (2011). DCI Glossary, available at http://folk.uio.no/trygver/2011/DCI-Glossary.pdf, last visited Jan. 1 2014.

Reenskaug, T. and Coplien, J.O. (2009). The DCI Architecture: A New Vision of Object-Oriented Programming, available at http://www.artima.com/articles/dci_vision.html, last visited Jan. 1 2014.

Roger, R. (2003). The real cost of aspect-oriented programming. *J. IEEE Software*, Vol. 20, No.6, pp 92–93.

Schärli, N., Ducasse, S., Nierstrasz, O. and Black, A. (2003). Traits: Composable Units of Behaviour. ECOOP 2003 – Object-Oriented Programming, ISBN 978-3-540-40531-3, pp. 248–274, Springer Verlag.

Stephen, R.P. and Felsing, J.M. (2001). A Practical Guide to Feature-Driven Development. Ed.1, ISBN 0130676152, Pearson Education.

Szyperski, C. (2002). Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Boston, MA, 2nd edition.

Thaker, S., Batory, D., Kitchin, D. and Cook, W. (2007). Safe Composition of Product Lines. *International Conference Generative Programming and Component Engineering (GPCE)*. ACM Press, pp. 95–104.

Viega, J. and Voas, J. (2000). Can Aspect-Oriented Programming Lead to More Reliable Software. *IEEE Software*, Vol. 17, No. 6, pp. 19–21.

Wirth, N. (1971). Program development by stepwise refinement. *Commun. ACM*, Vol. 14, No. 4, pp. 221–227.

Williams B. J. (2010), Carver J. C., Characterizing software architecture changes: A systematic review, *Information and Software Technology*, Vol. 52, No. 1, pp. 31–51.

Zagal, J.P., Ahue´s, R.S. and Voehl, M.N. (2002). Maintenance-Oriented Design and Development: A Case Study. *IEEE Software*, Vol. 19, No. 4, pp. 100–106.

Filman, R. et al. (2004). Aspect-Oriented Software Development, ISBN 0321219767, Addison-Wesley Professional.

Zhang, C. and Jacobsen, H.A. (2003). Refactoring Middleware with Aspects. *IEEE Trans. Parallel and Distributed Systems*, Vol. 14, No. 11, pp. 1058–1073.